Project 2:

Title:
Checkers

Class:
CSC 17C - 48295

Due Date:
December 4th, 2022

Author:
Aamir Khan

Introduction and Gameplay:

Title: Checkers

Objective:

The objective of this game is to capture all of your opponent's pieces.

Rules:

Checkers is a 2-player game that is played on an 8 x 8 board. Each player will be given 12 pieces. These pieces can be either red or black. The pieces will be placed on the board in a certain manner. Pieces can only move diagonally forward. You are NOT allowed to move your pieces vertically or horizontally. The goal of the game is to capture all of your opponent's game pieces. In the event that you capture all of your opponent's pieces, you will be announced the winner of the round. In order to capture pieces, you have to "jump" opponent pieces. To jump a piece, the square after the piece you'll be capturing must be empty. You will move your piece into the square that is after the piece you'll be capturing and take your opponent's piece. Moreover, if you are able to jump multiple pieces, you must jump all capturable pieces during your turn. If you can capture more than one piece, you'll be given the option to choose which piece you'd like to capture. If a piece reaches the opposite end of the board, it becomes a King piece that is capable of moving diagonally forward and backward. King pieces are allowed to move in either direction. Capturing a King piece is the same as capturing regular pieces. If, during your turn, you have no legal moves to make, you will forfeit the game. If this is the case, your opponent will receive all of your pieces, hence, winning the game.

Overview:

This project is a direct extension of my first checkers project. In it, I've added some new concepts that I've recently gone over. The concepts that I have added are: recursion, recursive sorts, hashing, and trees and graphs. The line count has increased substantially due to the inclusion of all of the concepts mentioned above, but the game's functionality hasn't changed since the first project. The intent for this project was to familiarize myself with some more advanced topics and find ways to implement such concepts within my project. Having completed this project, I now have a better understanding of all of the concepts I've included, such as how trees behave and how recursive sorts break down tasks into subtasks. I took this project on as a learning opportunity to see how different concepts play different roles in my project. Moreover, I can now begin to implement these concepts elsewhere and think about when these concepts would be useful and beneficial over others.

Concepts Included:

1. Container Classes:
    a. Sequential Containers:
        i. Lists - Used to create my board. (Found in my Board.h and .cpp files.)
    b. Associative Containers:
        i. Sets - Used in my Board.cpp file. (Found in my functions that test whether we can capture any pieces.)
        ii. Maps - Used in Main and in my Board.cpp file. (Found in Main as a way to keep count of how many captured pieces each player has.)
    c. Container Adaptors:
        i. Queues - Used in Main. (Found in my function that is used to display game rules. This is a simple use of a queue in my program to display the rules of checkers.)

2. Iterators:
    a. Input Iterators - Used in my Board.cpp file. (Used to fill my board (2-D list) with pieces. I also use input iterators to set the properties of the piece objects.)
    b. Output Iterators - Used in my Board.cpp file. (Used to display my piece objects.)
    c. Forward Iterators - Used in my Board.cpp file. (Used to traverse my board object and handle board operations. An example of a board operation using a forward iterator is found in my jumpTo() function.)
3. Algorithms:
    a. Non-Mutating Algorithms:
        i. find() - Used in my functions that check whether we can capture pieces. (I use the find function as a way to validate input and guarantee that an element from within my set is chosen.)
    b. Mutating Algorithms:
        i. swap() - Used to swap positions of pieces. (Used in my Piece.h file.)
    c. Organizing Algorithms:
        i. min() - Used in Main. (Used to determine the minimum number of pieces captured.)
        ii. max() - Used in Main. (Used to determine the maximum number of pieces captured.)
4. Advanced Topics:
    a. Recursion - Used to display King pieces in descending order through a recursive sort and also used to display the total time elapsed as a fraction of minutes through a recursive gcd function.
    b. Hashing - Used to check the likelihood of a random piece having been captured by either player. A random piece between 1-12 is chosen, and hashing is used to determine whether either player has captured the given piece. (Found in Main.)
    c. Trees & Graphs - Used to hold the pieces each player has captured. After every capture, the captured piece is stored in a tree that will be traversed at the end of our game. At the end, we display all of the pieces we've captured. (Found in Main.)

Stages of Development:

Version 1: Board Creation

In version 1, I began my project by creating an empty board to represent a checkers board. I then went on to display this empty board. I kept this version as simple as possible, so that we could see how the board would be developed from scratch.

Version 2: Board Basics

In version 2, I began adding on to our empty board. I started by generating the checkerboard pattern on our board. Squares that were marked with X's indicated squares that our pieces couldn't move into. Then, I added functions to handle and manipulate our board. One of the functions I included lets us jump to a specific square on our board. This will eventually be used to jump to a piece we'd like to move.

Version 3: Board Class

In version 3, I turned my checkers board into a class. The class contains the functions that I wrote in the previous version. Moreover, I demonstrated my board class in Main by creating a board object and using our jumpTo() function to jump to a specific square on our board. Then, I changed the contents of the square we jumped to.

Version 4: Placing Pieces

In version 4, I start placing pieces onto our checkers board. An "R" indicates a red piece, and a "B" indicates a black piece on our board. I place red pieces at the top of our board, and I place black pieces at the bottom of our board. I also added a function to check whether we have a valid square. This function takes an x- and y- coordinate and returns true if the square these coordinates represent are in our board and not marked by an X. Otherwise, it returns false.

Version 5: Piece Class

In version 5, I turn my pieces into a piece class. Each piece has a string member to indicate whether it's a black or red piece and a pair member to represent its position. I turn my 2-D list into a 2-D list of Piece * objects. This list used to be a 2-D list of strings, but now each element in our list represents a piece. Squares that are empty on our board are understood to be empty pieces. In other words, these are not actual pieces that can be used. When we eventually start moving our pieces, we can set the properties of an empty square to match our piece's properties. This can have the intended effect of our piece moving across the board.

Version 6: Piece Movement Part 1

In version 6, I added a function to our board class that will allow us to move a piece on our board. Within this function, I ask the user what piece they'd like to move and where they'd like to move the chosen piece. If all inputs are valid, a valid move is executed. When making a move, I needed to ensure that the player chose to move diagonally and in the right direction. A great deal of this move function incorporated input validation to guarantee that we chose valid inputs.

Version 7: Piece Movement Part 2

In version 7, I spend time working out how to choose valid pieces and move them. I check whether a piece we've chosen can even move by checking its diagonals and seeing whether the squares are empty. If a piece we've chosen can't move at all, I prompt the player to choose a different piece to move. I also ensure that a piece moves diagonally to the right or left. A piece should not be able to jump or skip any squares. Red pieces must move down on the board, and black pieces must move up on the board.

Version 8: Piece Movement Part 3

In version 8, I introduce the most important functionality and logic into my game, capturing pieces. If a piece can jump an opponent piece, I make sure that the opponent piece is removed from the board and that our piece moves into the square directly after. When jumping pieces, the square after the opponent piece must be empty. If an opponent piece is in your way but the square after it isn't empty, you are not able to jump the opponent piece. This program only allows us to capture one piece at a time, so we'll need to add functionality to capture multiple pieces.

Version 9: Piece Movement Part 4

In version 9, I add functionality to capture multiple pieces if the option is available. This version builds on version 8 as it allows us to capture all possible pieces that can be captured. If a player uses their piece to jump an opponent piece, we check to see if the player can jump any other pieces. As long as the player can jump a piece, they have to jump opponent pieces. If the player can capture more than one piece, they are given the option to choose which piece they'd like to capture. With this functionality added, a player now has to capture all possible pieces that are in their way.

Version 10: Piece Movement Part 5

In version 10, I add in King pieces. These are pieces that are capable of moving in both directions. If a piece reaches the opposite end of the board, it becomes a King piece. King pieces are captured in the same way regular pieces are. I also add a function to test whether we need to forfeit the game. In this function, we check every one of our pieces to see if we have at least one piece with a legal move. If we have at least one piece with a legal move, we don't forfeit the game. However, if we have no pieces with a legal move, we have to forfeit the game.

Version 11: Adding Players

In version 11, I add players into our game. Players are able to choose whether they'd like to play with black pieces or red pieces. Additionally, I utilize a map to keep count of how many pieces each player has captured. The map uses a player's piece as its key and it stores the number of pieces the given player has captured as its value. The idea behind the use of this map is to have a way to keep count of how many pieces both players have captured. Once a player has captured all 12 opponent pieces, we can end the game and announce the winner.

Version 12: Adding Algorithms

In version 12, I made some minor additions to our game. I added some simple STL algorithms into my game such as the min and max functions. I also used the set's built-in find function as a way to validate input. In another area of my program, I use the STL's swap algorithm to swap the positions of pieces. This version doesn't have any major changes besides the addition of the algorithms mentioned above. At the end of Main, I display the minimum and maximum number of pieces captured throughout all turns.

Version 13: Start Game

In this version, a full game of checkers can be played with two players. The game ends when either player captures all 12 opponent pieces. At the end, the maximum and minimum number of pieces captured are displayed. The time taken for the game is also displayed. Lastly, the winner is announced and the game comes to a close.

Version 14: Adding Trees

In this version, I added trees into my program. I used trees to hold the pieces each player captured. Whenever a player captured a piece, I would add that captured piece to the player's tree data structure. At the end of the game, I would display the pieces each player captured by

traversing their tree using an in-order traversal. One thing to note is that the tree data structure I implemented was an AVL tree. It is a self-balancing binary tree that still upholds all properties of binary trees.

Version 15: Adding Hashing

In this version, I added hashing into my program. I used hashing as a way to test whether a random piece was likely captured by either player at the end of the game. After every player's turn, I would add their count of captured pieces into a hash table using a hash value generated with their name. Once the game came to an end, I would choose a random piece between 1-12 to look for. I would then go through all of the values that were hashed for each player to look for whether the chosen piece was inserted at some point during the game. If the piece was found, I would output that the piece was likely captured. Otherwise, I would output that the piece wasn't found. There can be collisions, so our result is only probabilistic.

Version 16: Adding Recursion

This is the final version of my project. In this version, I added some recursion into my project. I used a recursive gcd function to display the time elapsed as a fraction of minutes. I also used a recursive quick sort algorithm to display King pieces in descending order. Recursion involves breaking down large tasks into smaller subtasks, so I wanted to better understand how recursion could be beneficial in simplifying logic within my project. I implemented recursion in only 2 areas of my project. In the future, I could use recursion within my game to accomplish much more.

Checkers Pseudo-Code:

```
/*
 * File:       main.cpp
 * Author:   Aamir Khan
 * Created:  November 20, 2022, 10:30 AM
 * Purpose: Checkers Pseudo-Code.
 */

//System Libraries
//Fstream Library.
//Queue Library.
//Namespace Std.

//User Libraries
//Player Class.

//Function Prototypes
//List my function prototypes.

//Main -> Execution Begins Here:
        //Ask the user if they'd like to see the rules. If they would, we display the rules.

        //Create a board object that will be played on.

        //Create 2 player objects and set them up.

        //Use trees to hold the pieces each player has captured throughout the game.
        //Use a map to keep count of the number of captured pieces of both players.
        //Use a hash table to hash the pieces each player has captured. (Searched in the end.)

        //As long as neither player has captured all 12 opponent pieces…
                //We keep taking turns in moving our pieces.
                //If we have no legal moves to make, we forfeit the match.

        //Once a player has captured all 12 opponent pieces, we display some information:

        //We output how long the game took, the number of pieces captured by both players,
        //whether a randomly chosen piece was captured by either player, and the winner.
        //(We search our hash table for a randomly chosen piece to see if it has been captured.)

        //Clean up - Deallocate the memory we used for our objects.

//Exit Main - End of the Program.
```

Sample Screen Shots:

```
//As long as neither player has captured all 12 opponent pieces, we loop.
while(captPces[plyr1->getPiece()]!=allPces&&
      captPces[plyr2->getPiece()]!=allPces){
    //Making a move with our player.
    cout<<plyr1->getName()<<"'s Turn:\n";
    plyr1->makeMove(captPces,tree1);
    //After our turn, we store our count in the hash table.
    int indx=hashFnc(plyr1->getName())%allPces;
    table[indx].push_back(captPces[plyr1->getPiece()]);
    cout<<endl;

    //Making a move with our opponent.
    cout<<plyr2->getName()<<"'s Turn:\n";
    plyr2->makeMove(captPces,tree2);
    //After the opponent's turn, we store their count in the hash table.
    indx=hashFnc(plyr2->getName())%allPces;
    table[indx].push_back(captPces[plyr2->getPiece()]);
    cout<<endl;

}
```

This is an image of the loop responsible for taking turns. It will continue to loop as long as neither player has captured all 12 opponent pieces.

```
cout<<"Closing Announcements:\n";
cout<<"--------------------------------------------------------------------------\n";
//Storing the maximum and minimum number of pieces captured.
int maxCapt=max(captPces[plyr1->getPiece()],captPces[plyr2->getPiece()]);
int minCapt=min(captPces[plyr1->getPiece()],captPces[plyr2->getPiece()]);
//Displaying the maximum and minimum number of pieces captured.
cout<<"Time elapsed = "<<tot/dnmtr<<"/"<<60/dnmtr<<" minutes.\n"<<endl;
cout<<"Maximum number of pieces captured = "<<maxCapt<<endl;
cout<<"Minimum number of pieces captured = "<<minCapt<<endl;
cout<<endl;
```

This is an image of some of the announcements and messages that are displayed after the game ends. We display the time elapsed as a fraction of minutes and the maximum and minimum number of pieces captured.

```
//Declaring 2 trees to hold the pieces each player has captured.
Tree *tree1=new Tree(); //For player 1.
Tree *tree2=new Tree(); //For player 2.

//Declaring a map to keep count of how many pieces each player has captured.
map<string,int> captPces;
//The total number of pieces each player begins with.
int allPces=12;

//Declaring a hash table that will be used to check whether a randomly
//chosen piece was possibly captured at the end of our game or not.
list<int> table[allPces]; //12 valid indices/lists.
```

Here is an image showing the use of trees, maps, and hashing. As seen below, we are creating a hash table that will eventually be searched at the end of our game.

```
//Displaying the pieces player 1 has captured.
cout<<plyr1->getName()<<" has captured the following pieces:\n";
//If player 1 has captured at least 1 piece, we display the pieces they've captured.
if(!tree1->empty()) tree1->prntIn();
//Otherwise, we display that they've captured none.
else cout<<"NONE\n";
//Displaying the pieces player 2 has captured.
cout<<plyr2->getName()<<" has captured the following pieces:\n";
//If player 2 has captured at least 1 piece, we display the pieces they've captured.
if(!tree2->empty()) tree2->prntIn();
//Otherwise, we display that they've captured none.
else cout<<"NONE\n";
```

Here is an image of us displaying the pieces captured by both players. We used trees to hold the pieces captured by each player, so we displayed the trees of both players at the end. If a tree is empty, we simply display that no pieces were captured by the given player.

```
//Searches a list for a value and returns whether it was found or not.
bool find(list<int> l,int target){
    //Looking through every element of the list.
    for(int val:l){
        //If we've found our value, we return true.
        if(val==target) return true;
    }
    //Not found...
    return false;
}
```

Here is an image of a function that will be used to look through our hash table for a specific value. It searches through a list for a value and returns whether it was found or not.

```
//Checking to see if a random piece was possibly captured.
cout<<"\nChecking to see if a random piece was possibly captured:\n";
cout<<"-----------------------------------------------------------------\n";
int randPce=rand()%allPces+1; //A random piece between 1-12.
cout<<"Random piece to check = "<<randPce<<endl;
//Getting player 1's hash value.
int indx=hashFnc(plyr1->getName())%allPces;
//Looking through our hashed list to see if we might've captured the piece.
bool found=find(table[indx],randPce);
found?cout<<plyr1->getName()<<" might've captured piece "<<randPce<<"."<<endl:
      cout<<plyr1->getName()<<" did not capture piece "<<randPce<<"."<<endl;
//Getting player 2's hash value.
indx=hashFnc(plyr2->getName())%allPces;
//Looking through our hashed list to see if we might've captured the piece.
found=find(table[indx],randPce);
found?cout<<plyr2->getName()<<" might've captured piece "<<randPce<<"."<<endl:
      cout<<plyr2->getName()<<" did not capture piece "<<randPce<<"."<<endl;
```

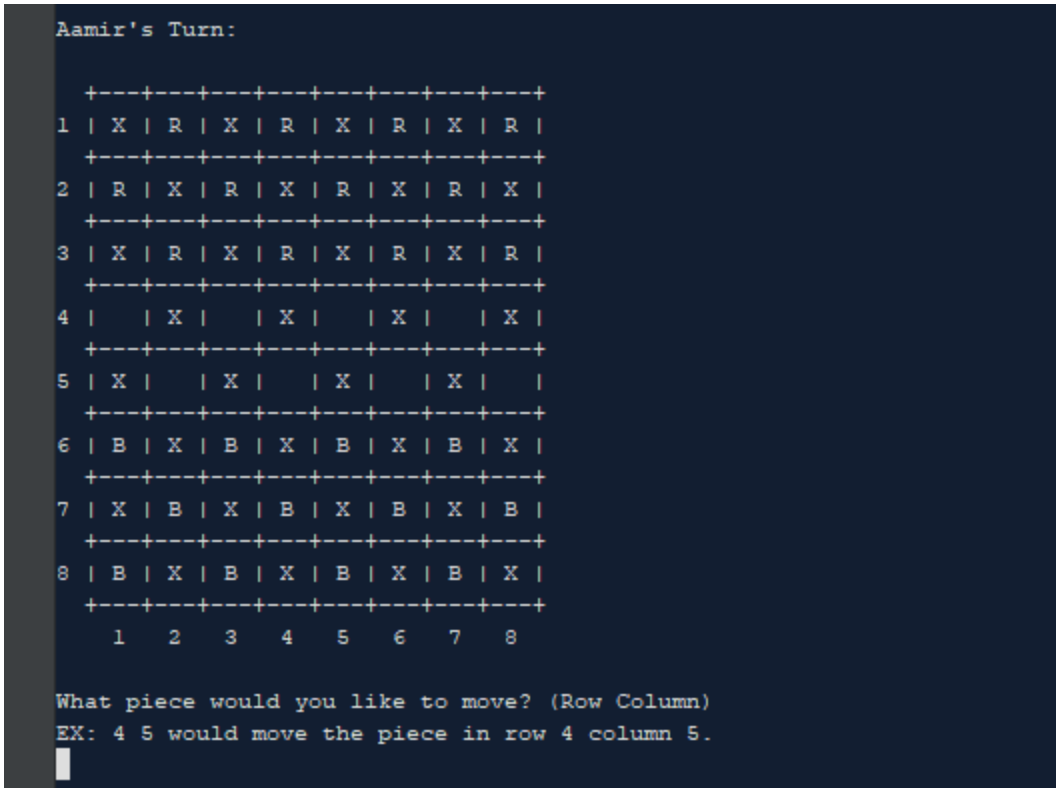
This is an image of us searching our hash table for a randomly chosen piece. We search through the hashed lists of both players for the chosen value. If the value is found, we display that the value was likely captured.

```
/** In-Order traversal. */
void Tree::prntIn(Node *root){
    //Base case:
    if(root==NULL) return;
    //Left->Root->Right:
    prntIn(root->left);
    cout<<root->data<<" ";
    prntIn(root->right);
}
```

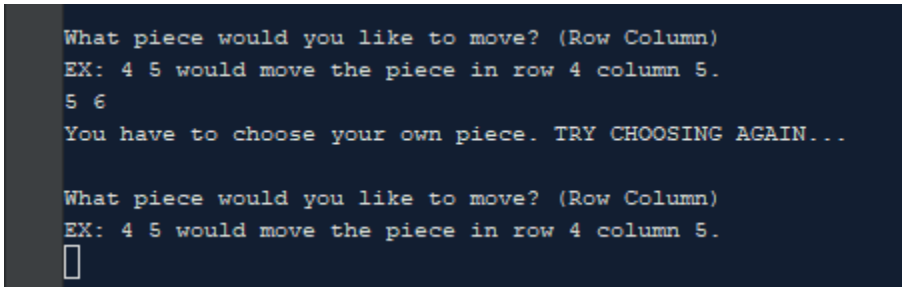
This is an image of an in-order traversal function that is utilized by our tree class.

```
/** \brief A node struct. to represent nodes in a tree.*/
struct Node{
    int data;    /**< Our data. */
    Node *left;  /**< Our left child. */
    Node *right; /**< Our right child. */
};
```

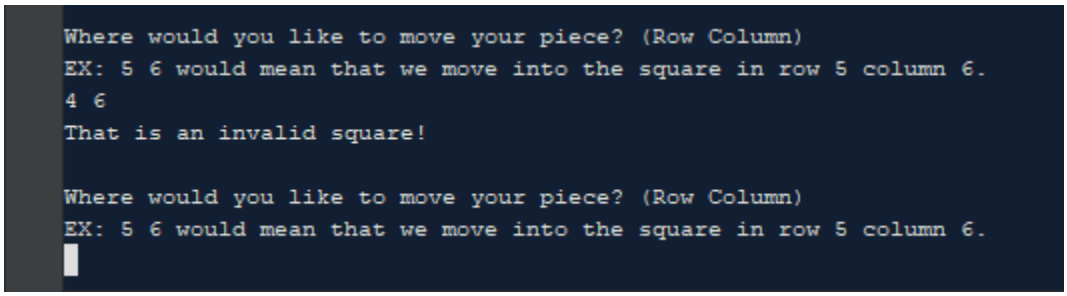
This is an image of our Node struct that is used within our tree class. It is a struct that represents a node in a tree. We can see that it is self-referential as it has 2 members of the same type as itself.



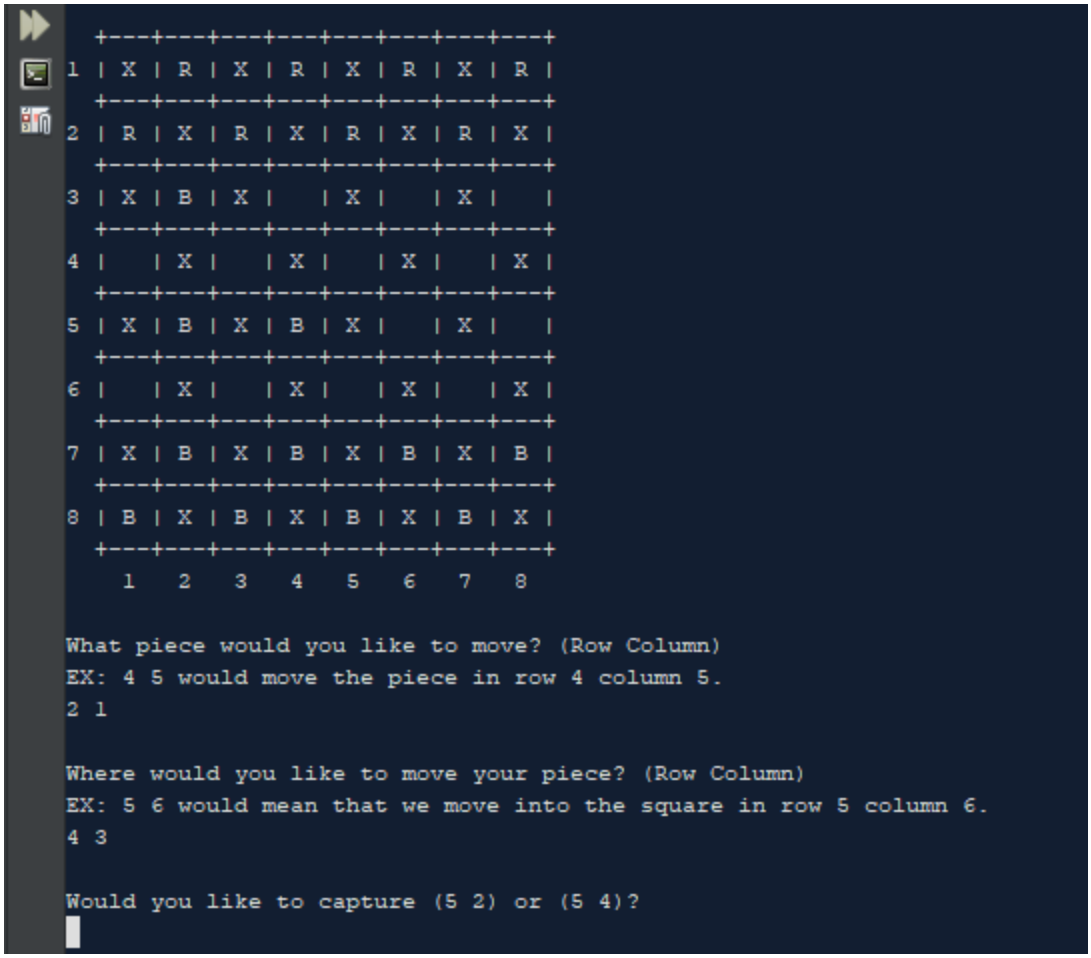
This is an image of a player being prompted for their turn. It also showcases our checkers board. To make it easier for players to identify the rows and columns, I number them.



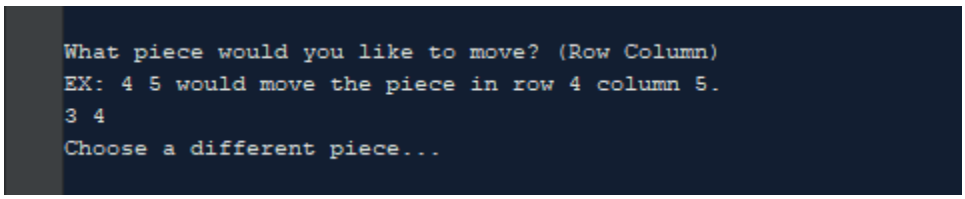
This is an error message a player receives if they choose a piece that isn't their own. They will then be prompted again to choose their own piece.



This is an error message that is displayed if a player chooses a square that's marked with an X.



This is an image that demonstrates our ability to choose between pieces to capture. I first used the red piece in R2 C1 to capture the black piece in R2 C3 by jumping into R4 C3. Then, I was given the option to choose what black piece in R5 I would like to capture. After choosing a piece, I would then need to jump into the square directly after the chosen piece to capture.



This is an error message that is displayed if we choose a piece that can't move. If there are no legal moves to make with our chosen piece, this message will display and we'll be prompted to choose a different piece.

```
You have to move down on the board...
You can only move to the left/right diagonally.
Where would you like to move your piece? (Row Column)
```

This is an error message that is displayed if we use a red piece to move up the board. Red pieces are placed at the top, so they are only allowed to move down the board. Only King pieces are allowed to move in either direction.

```
This piece is now a King piece!

  +---+---+---+---+---+---+---+---+
1 | X | R | X | R | X | R | X | R |
  +---+---+---+---+---+---+---+---+
2 | R | X | R | X | R | X | R | X |
  +---+---+---+---+---+---+---+---+
3 | X |   | X | B | X | R | X |   |
  +---+---+---+---+---+---+---+---+
4 |   | X |   | X |   | X |   | X |
  +---+---+---+---+---+---+---+---+
5 | X |   | X | B | X |   | X |   |
  +---+---+---+---+---+---+---+---+
6 |   | X | B | X | R | X | B | X |
  +---+---+---+---+---+---+---+---+
7 | X | B | X |   | X | B | X | B |
  +---+---+---+---+---+---+---+---+
8 | B | X | R | X | B | X | B | X |
  +---+---+---+---+---+---+---+---+
    1   2   3   4   5   6   7   8
```

This is an image of a red piece becoming a King piece. When pieces reach the opposite end of the board, they become King pieces that are capable of moving in both directions. The square in R8 C3 is being occupied by a red piece, so this red piece becomes a King piece.

```
Here are your King pieces...
(8 3)

What piece would you like to move? (Row Column)
EX: 4 5 would move the piece in row 4 column 5.
```

At the start of every turn, a player's King pieces are displayed. Moreover, they are displayed in descending order. This helps us keep track of what pieces are King pieces.

```
/** \brief Our constructor. */
Board::Board(){
    //Creates an empty checkers board.
    this->fillBrd();
    //Places pieces onto our board.
    this->putPieces();
}

/** \brief Our destructor. */
Board::~Board(){
    //We declare an iterator to span our 2-D list.
    list<list<Piece *>>::iterator it;
    //We iterate over our 2-D list and destroy each piece.
    for(it=board.begin();it!=board.end();it++){
        //We declare an iterator to work with each list in our board.
        list<Piece *>::iterator itr;
        //Here, it points to each individual list.
        for(itr=it->begin();itr!=it->end();itr++){
            //Deleting each piece;
            delete *itr;
        }
    }
}
```

This is an image of my constructor and destructor for the board class. The constructor sets our board up. The destructor cleans up by deallocating the memory used in creating our board of Piece * objects.

```
/**To quick-sort a vector of pairs in descending order based on their
 * first values. We will use this sort to display our King pieces
 * in descending order. We input our vector of pairs, the starting
 * index of our vector, and the ending index of our vector.
 */
void Board::quickSort(vector<pair<int,int>> &v,int start,int end){
    //Base case:
    if(start>=end) return;
    //Partitioning our vector.
    int indx=partition(v, start, end);
    //Recursively quick-sorting our partitions.
    quickSort(v,start,indx-1); //Our left partition.
    quickSort(v,indx+1,end);   //Our right partition.
}
```

This is an image of the quick-sort algorithm that is used to display our King pieces in descending order. It is a recursive algorithm, so we see it calling itself.